



CMSC 5743

Efficient Computing of Deep Neural Networks

Lecture 02: Convolution Speedup

Bei Yu

CSE Department, CUHK

byu@cse.cuhk.edu.hk

(Latest update: September 22, 2021)

Fall 2021



These slides contain/adapt materials developed by

- Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*
- Asit K. Mishra et al. (2017). “Fine-grained accelerators for sparse machine learning workloads”. In: *Proc. ASPDAC*, pp. 635–640
- Jongsoo Park et al. (2017). “Faster CNNs with direct sparse convolutions and guided pruning”. In: *Proc. ICLR*
- UC Berkeley EE290: “Hardware for Machine Learning”
<https://inst.eecs.berkeley.edu/~ee290-2/sp20/>



① Convolution Basis

② GEMM

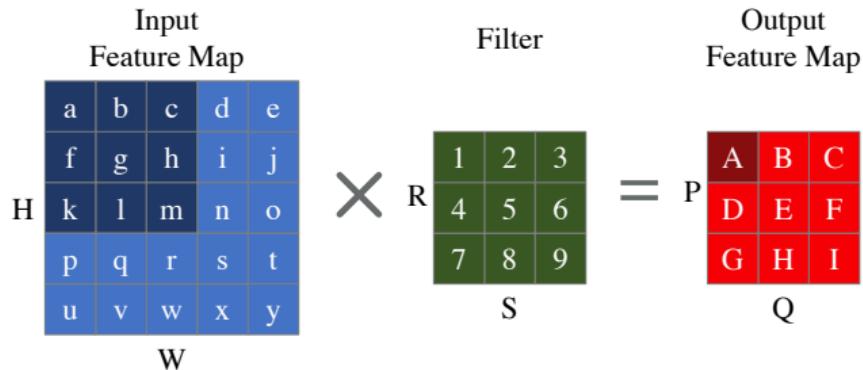
③ Direct Convolution

④ Sparse Convolution



Convolution Basis

2D-Convolution



- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map

$$A = a \cdot 1 + b \cdot 2 + c \cdot 3 \\ + f \cdot 4 + g \cdot 5 + h \cdot 6 \\ + k \cdot 7 + l \cdot 8 + m \cdot 9$$

2D-Convolution



$$\begin{array}{c} \text{Input} \\ \text{Feature Map} \\ \hline \begin{matrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{matrix} \\ H \quad W \end{array} \times R = \begin{array}{c} \text{Filter} \\ \hline \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \\ S \end{array} = \begin{array}{c} \text{Output} \\ \text{Feature Map} \\ \hline \begin{matrix} A & B & C \\ D & E & F \\ G & H & I \end{matrix} \\ P \quad Q \end{array}$$

- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- **stride**: # of rows/columns traversed per step

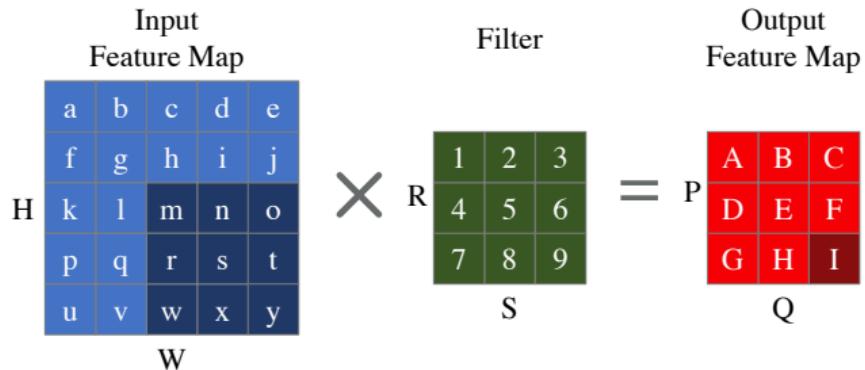
2D-Convolution



$$\begin{array}{c} \text{Input} \\ \text{Feature Map} \\ \hline \begin{matrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{matrix} \\ H \quad W \end{array} \times \begin{array}{c} \text{Filter} \\ \hline \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \\ R \quad S \end{array} = \begin{array}{c} \text{Output} \\ \text{Feature Map} \\ \hline \begin{matrix} A & B & C \\ D & E & F \\ G & H & I \end{matrix} \\ P \quad Q \end{array}$$

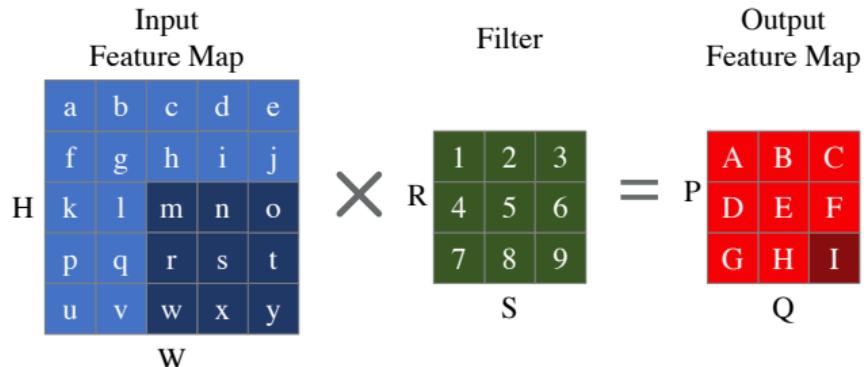
- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- **stride**: # of rows/columns traversed per step

2D-Convolution



- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- **stride**: # of rows/columns traversed per step

2D-Convolution

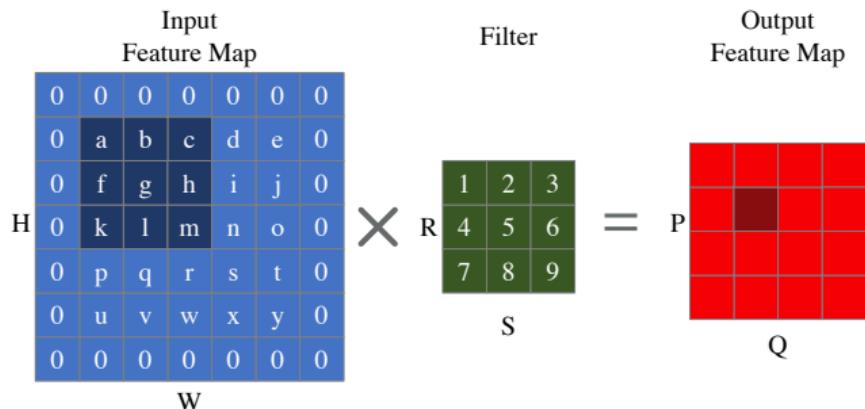


- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- **stride**: # of rows/columns traversed per step

$$P = \frac{(H - R)}{\text{stride}} + 1;$$

$$Q = \frac{(W - S)}{\text{stride}} + 1.$$

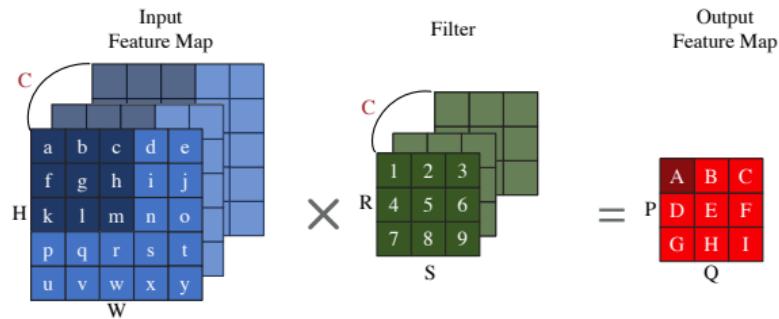
2D-Convolution



$$P = \frac{(H - R + 2 \cdot \text{pad})}{\text{stride}} + 1;$$
$$Q = \frac{(W - S + 2 \cdot \text{pad})}{\text{stride}} + 1.$$

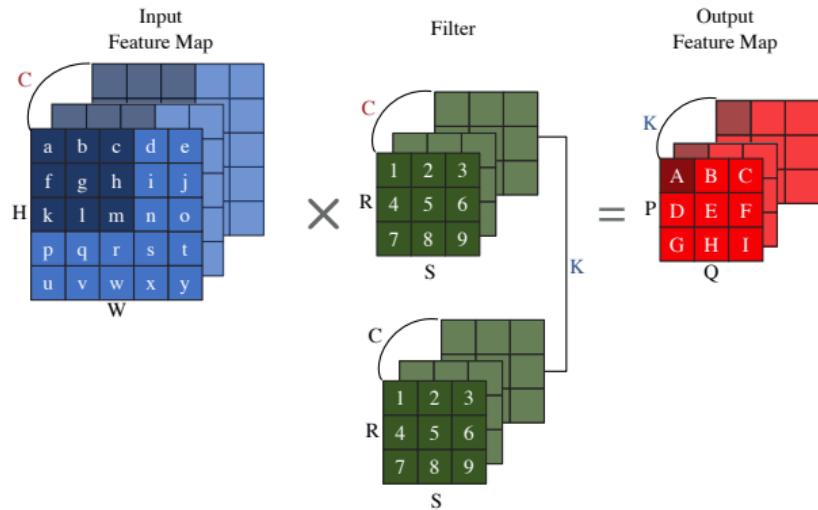
- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step
- padding: # of zero rows/columns added

3D-Convolution



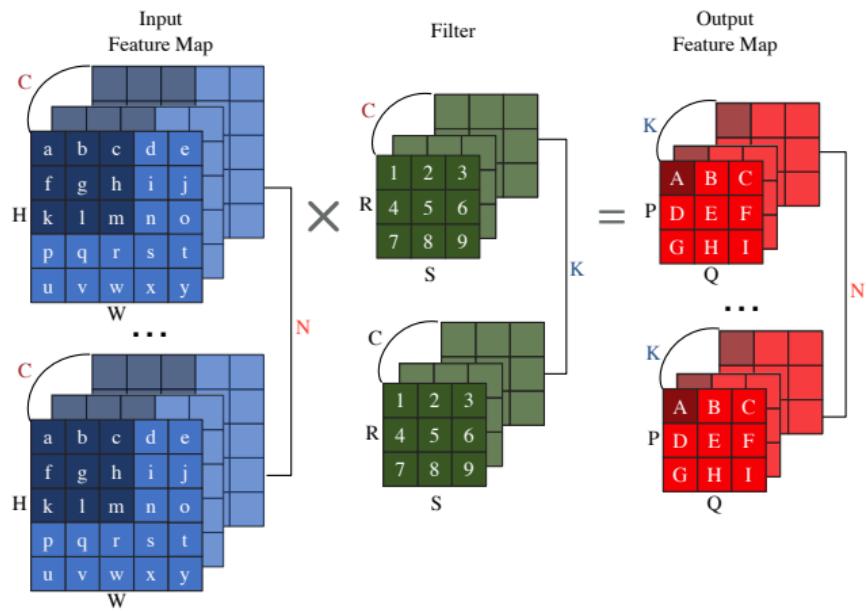
- H : Height of input feature map
- W : Width of input feature map
- R : Height of filter
- S : Width of filter
- P : Height of output feature map
- Q : Width of output feature map
- stride: # of rows/columns traversed per step
- padding: # of zero rows/columns added
- C : # of input channels

3D-Convolution



- H : Height of input feature map
- W : Width of input feature map
- R : Height of filter
- S : Width of filter
- P : Height of output feature map
- Q : Width of output feature map
- stride: # of rows/columns traversed per step
- padding: # of zero rows/columns added
- C : # of input channels
- K : # of output channels

3D-Convolution



- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step
- padding: # of zero rows/columns added
- C: # of input channels
- K: # of output channels
- N: Batch size

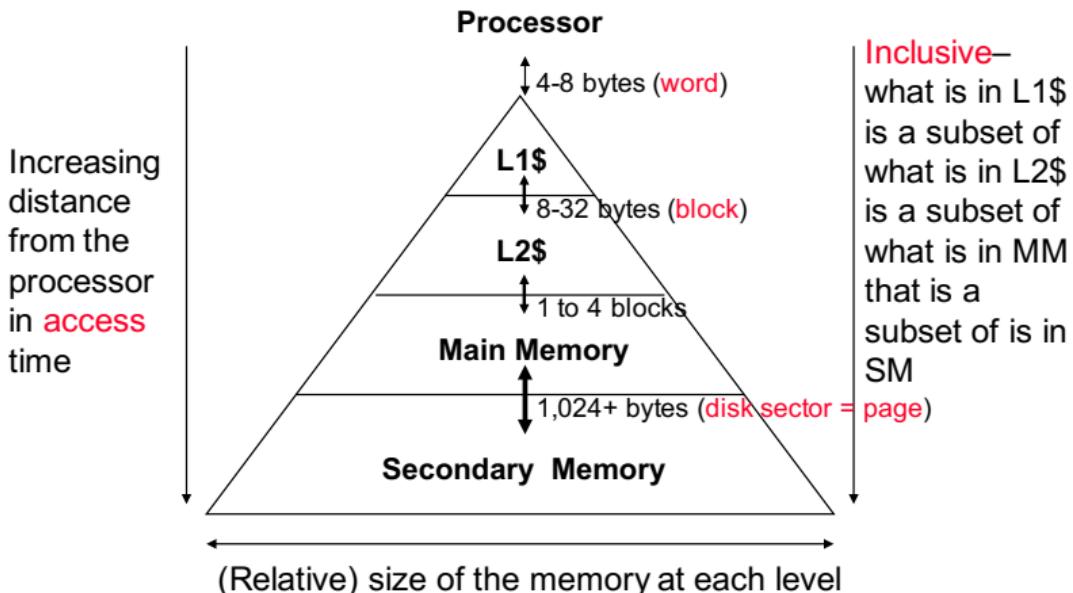


$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 2 & 2 & 1 & 1 & 2 & 0 & \\ \hline 0 & 2 & 0 & 1 & 1 & 0 & 0 & \\ \hline 0 & 2 & 0 & 1 & 2 & 0 & 0 & \\ \hline 0 & 1 & 1 & 1 & 1 & 1 & 0 & \\ \hline 0 & 0 & 0 & 1 & 0 & 2 & 0 & \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline 1 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline 1 & 0 & -1 \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|} \hline 4 & 6 & 3 & 5 & 4 \\ \hline 2 & 6 & 2 & 4 & 4 \\ \hline 1 & 5 & 3 & 4 & 4 \\ \hline 2 & 4 & 3 & 3 & 4 \\ \hline 0 & 2 & 2 & 4 & 3 \\ \hline \end{array}$$

Direct convolution: No extra memory overhead

- Low performance
- Poor memory access pattern due to geometry-specific constraint
- Relatively short dot product

Background: Memory System

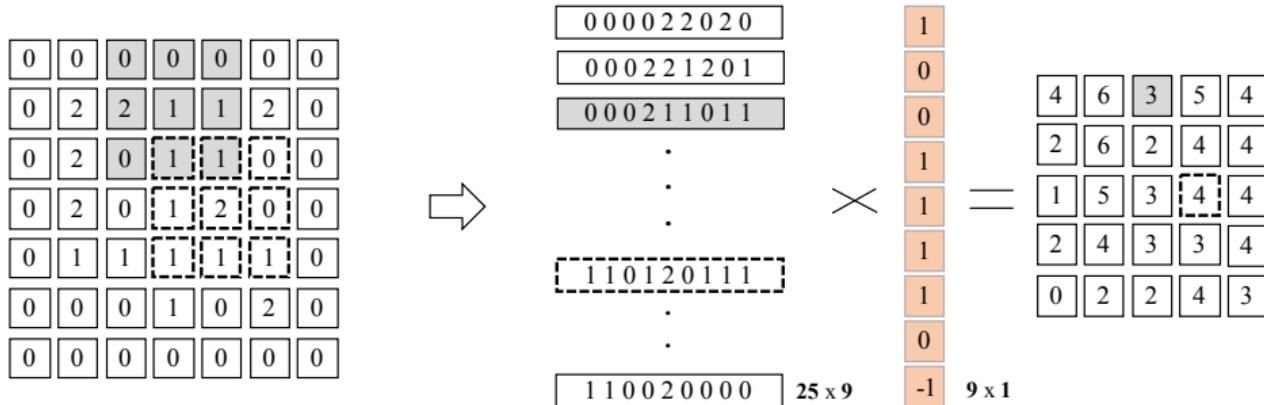


- **Spatial** locality
- **Temporal** Locality



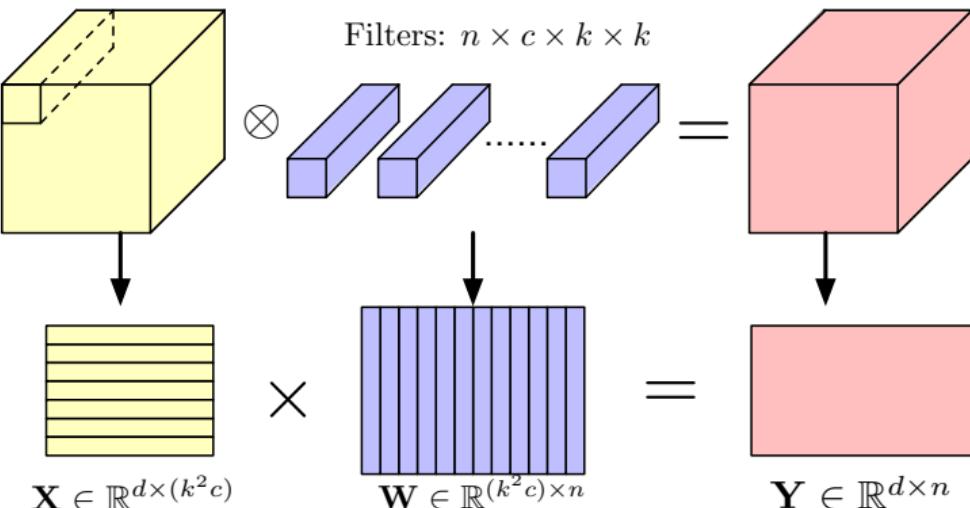
GEMM

Im2col (Image2Column) Convolution



- Large extra memory overhead
- Good performance
- BLAS-friendly memory layout to enjoy SIMD/locality/parallelism
- Applicable for any convolution configuration on any platform

Im2col (Image2Column) Convolution



- Transform convolution to matrix multiplication
- Unified calculation for both convolution and fully-connected layers

Im2col (Image2Column): Another View

Image to column operation (im2col)

Slide the input image like a convolution but each patch become a column vector.

Input Image [4x4x3]

	33	34	35	36
17	18	19	20	40
2	3	4	24	44
5	6	7	8	28

9	10	11	12	32
13	14	15	16	

Kernel Width:2
Kernel Height:2
Stride:1.
Padding:0

 $W_{out} = (W_{in} - kW + 2P)/S + 1$
 $H_{out} = (H_{in} - kh + 2P)/S + 1$
 $W_{out} = (4-2)/1+1=5$
 $H_{out} = (4-2)/1+1=5$

2x2x3 column vector
 $[1x4][1x3][1x2][1x3]$

Result: [12x9]

9 possible Sliding window positions

1	2	3	5	6	7	9	10	11
2	3	4	6	7	8	10	11	12
5	6	7	9	10	11	13	14	15
6	7	8	10	11	12	14	15	16
17	18	19	21	22	23	25	26	27
18	19	20	22	23	24	26	27	28
21	22	23	25	26	27	29	30	31
22	23	24	26	27	28	30	31	32
33	34	35	37	38	39	41	42	43
34	35	36	38	39	40	42	43	44
37	38	39	41	42	43	45	46	47
38	39	40	42	43	44	46	47	48

We get true performance gain
when the kernel has a large number of filters, ie: F=4
and/or you have a batch of images (N=4). Example for the input batch [4x4x3x4], convolved with 4 filters [2x2x3x2].
The only problem with this approach is the amount of memory

Reshaped kernel: [4x12]

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12

Converted input batch [12x56]

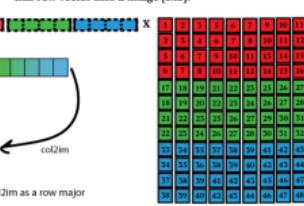
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48

We can multiply this result matrix [12x9]

result = kernel x matrix

The result would be a row vector [1x9].

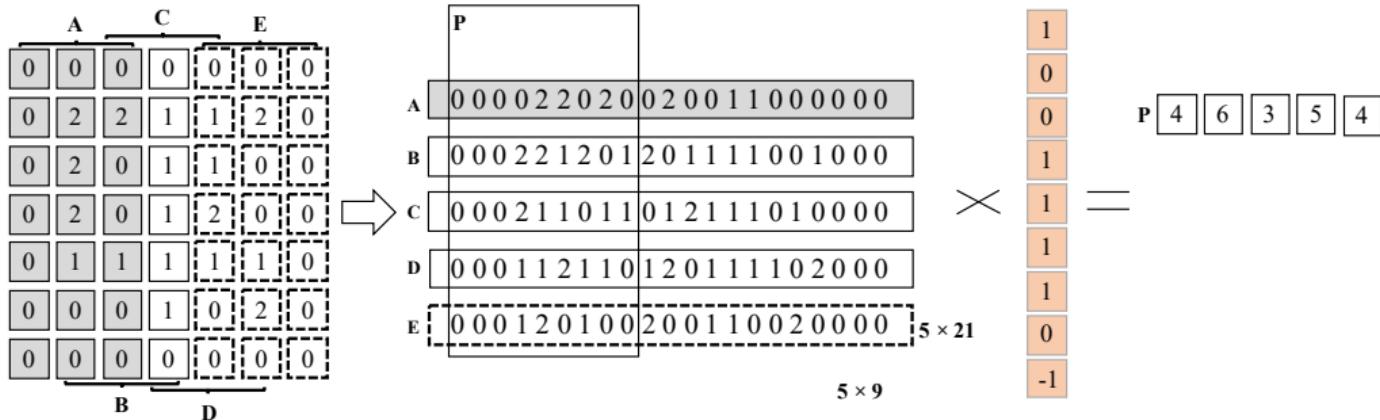
We need another operation that will convert this row vector into a image [3x3].



Consider col2im as a row major reshape.

Conv out: [1x3]

SOTA 1: Memory-efficient Convolution

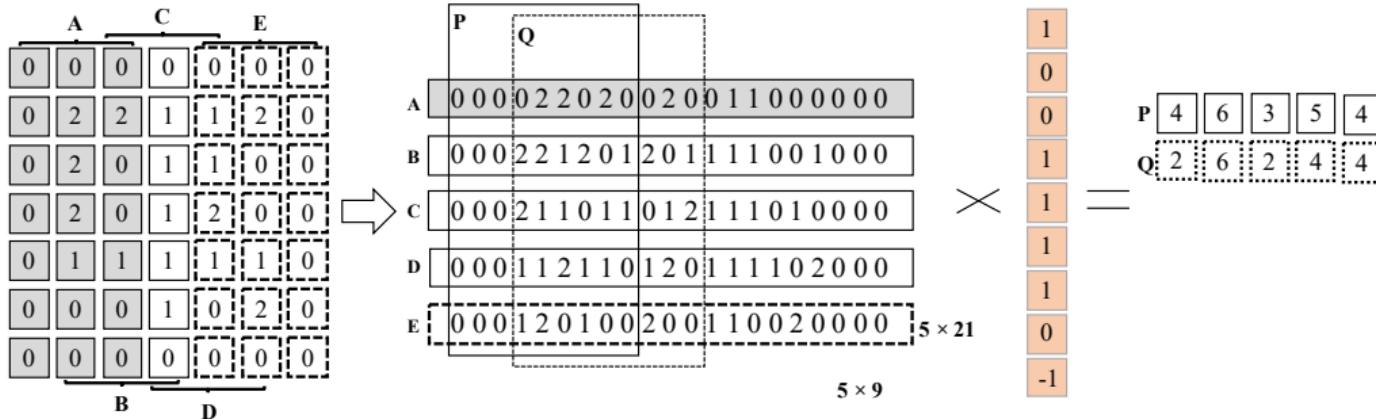


2

- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

²Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

SOTA 1: Memory-efficient Convolution

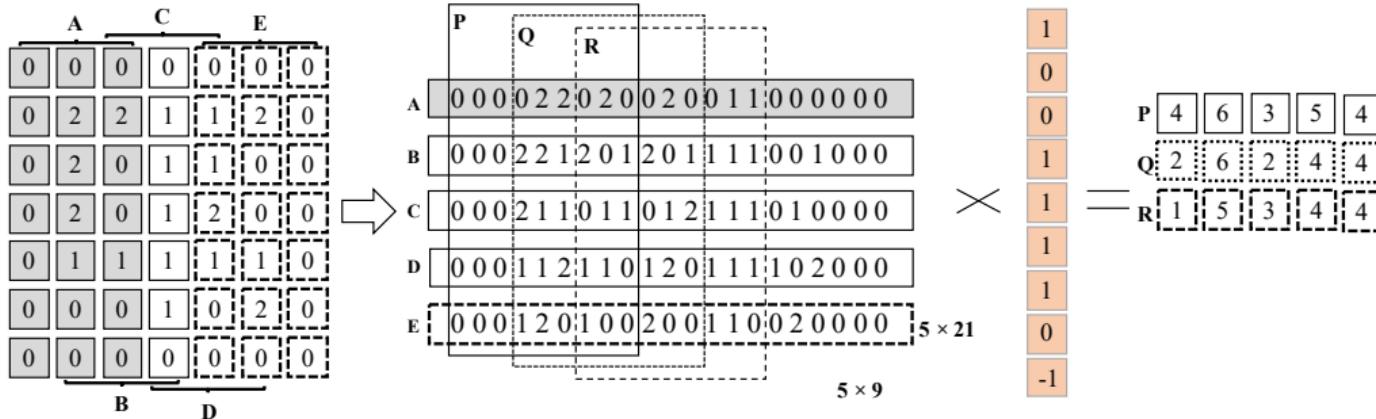


2

- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

²Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

SOTA 1: Memory-efficient Convolution

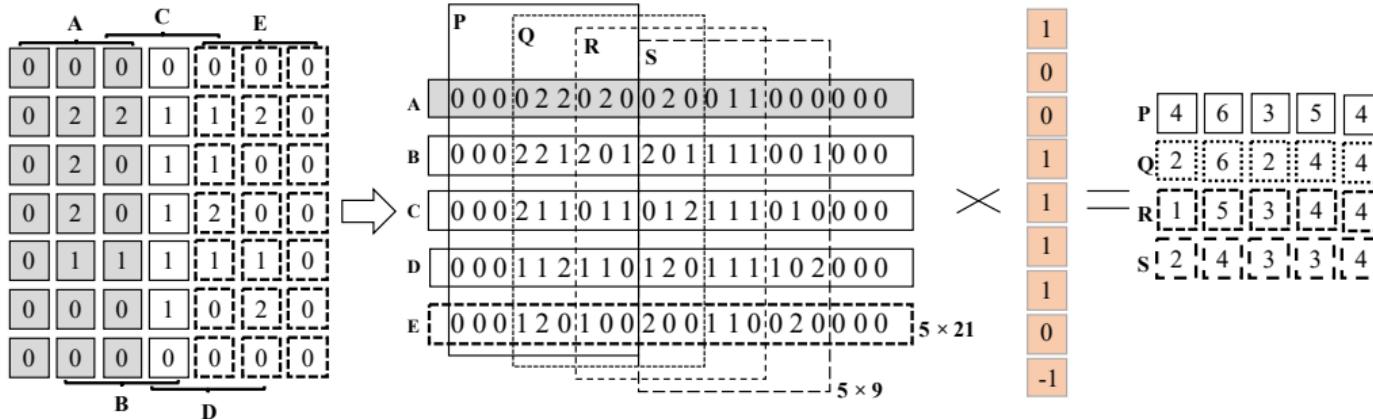


2

- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

²Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

SOTA 1: Memory-efficient Convolution

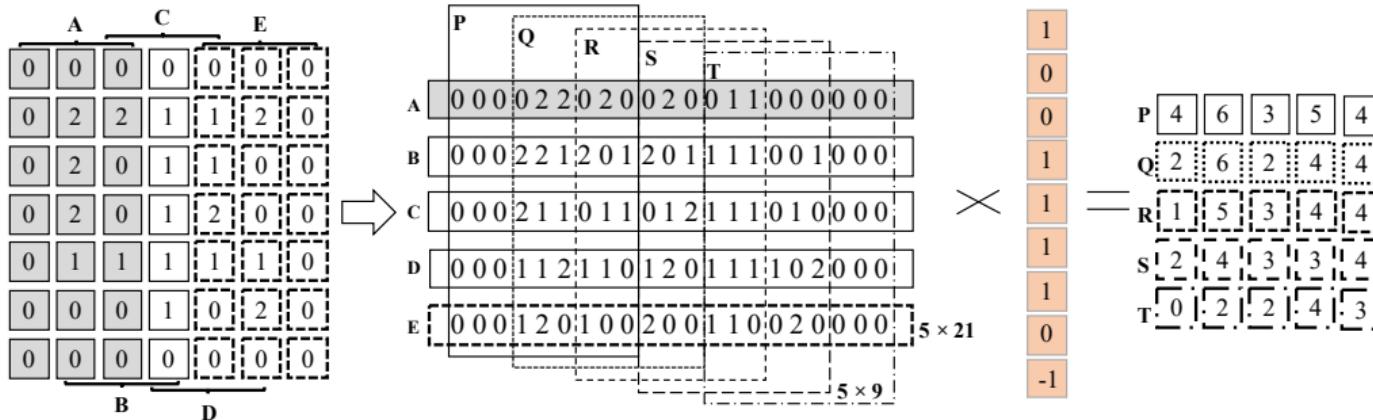


2

- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

²Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

SOTA 1: Memory-efficient Convolution



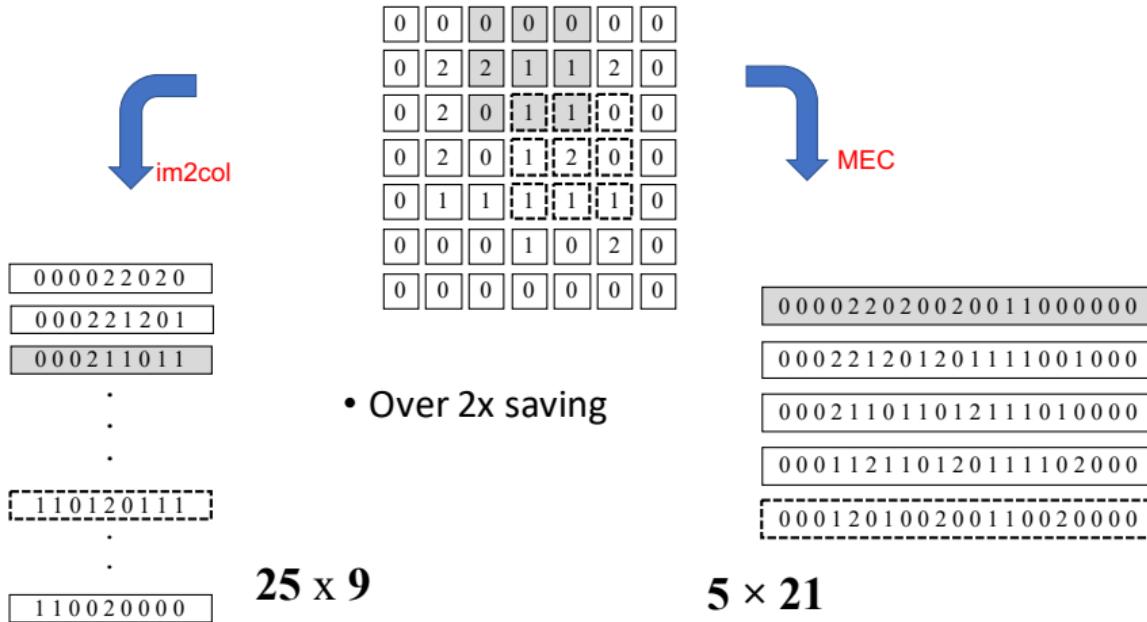
2

- Sub matrices in the lowered matrix will be “sgemm” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

²Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.



Over 2× memory saving³:

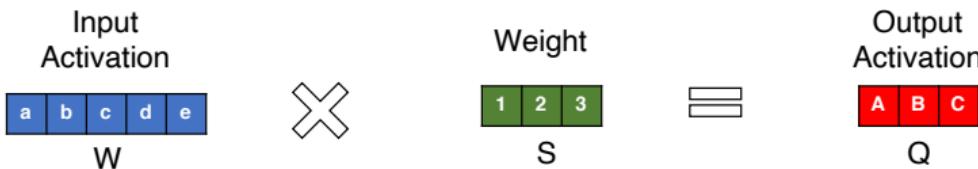


³Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.



Direct Convolution

1D Convolution Example



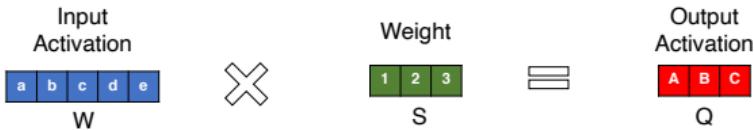
```
for(q=0; q<Q; q++) {  
    for (s=0; s<S; s++) {  
        OA[q] += IA[q+s] * W[s];  
    }  
}
```

**Output Stationary (OS)
Dataflow**

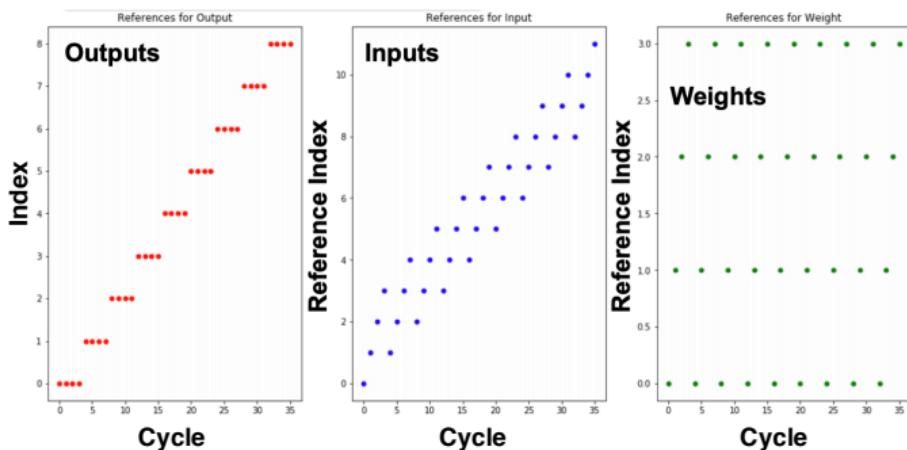
```
for (s=0; s<S; s++) {  
    for (q=0; q<Q; q++) {  
        OA[q] += IA[q+s] * W[s];  
    }  
}
```

**Weight Stationary (WS)
Dataflow**

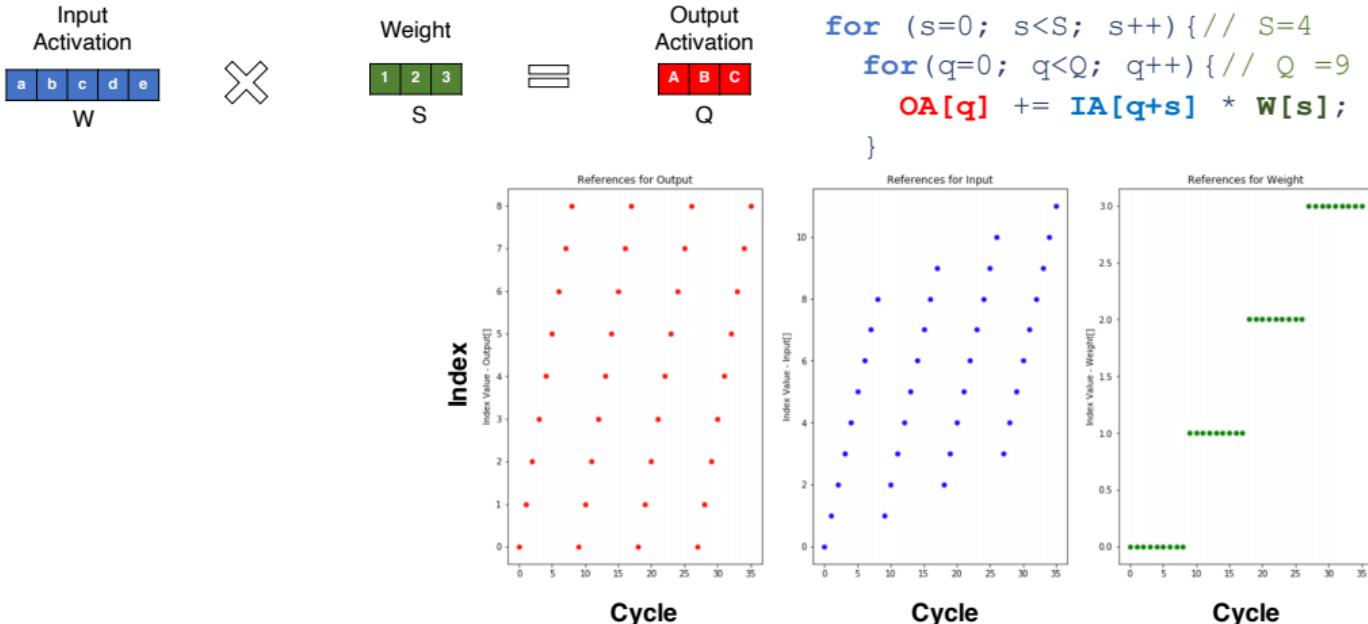
Buffer Access Pattern 1: Output Stationary



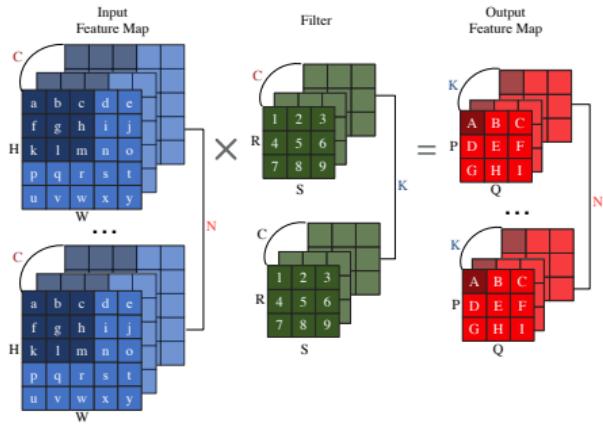
```
for (q=0; q<Q; q++) { // Q = 9
    for (s=0; s<S; s++) { // S=4
        OA[q] += IA[q+s] * W[s];
    }
}
```



Buffer Access Pattern 2: Weight Stationary

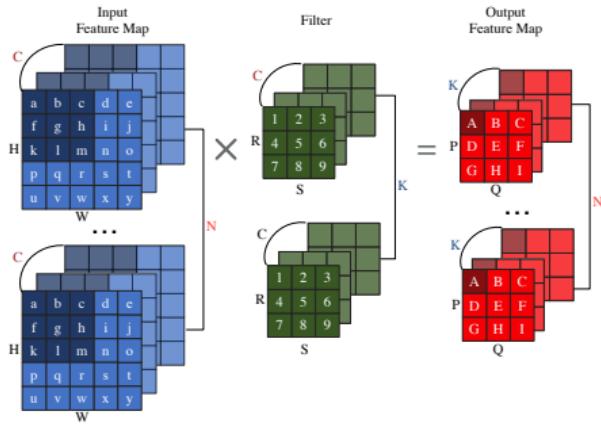


Direct Convolution



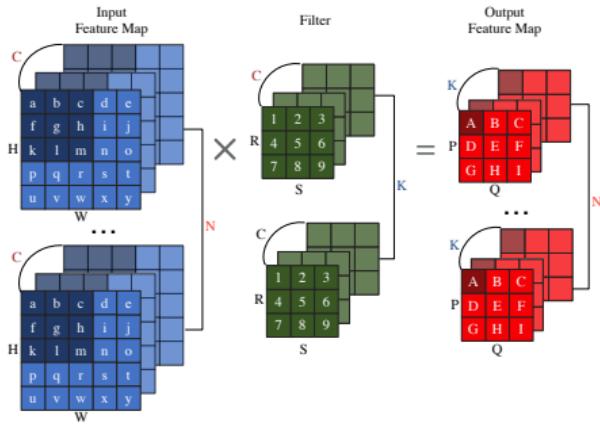
```
1   for (n=0; n<N; n++) {
2     for (k=0; k<K; k++) {
3       for (p=0; p<P; p++) {
4         for (q=0; q<Q; q++) {
5           OA[n][k][p][q] = 0;
6           for (r=0; r<R; r++) {
7             for (s=0; s<S; s++) {
8               for (c=0; c<C; c++) {
9                 h = p * stride - pad + r;
10                w = q * stride - pad + s;
11                OA[n][k][p][q] += IA[n][c][h][w] * W[k][c][r][s];
12             } } } } } }
```

Direct Convolution: Loop Ordering



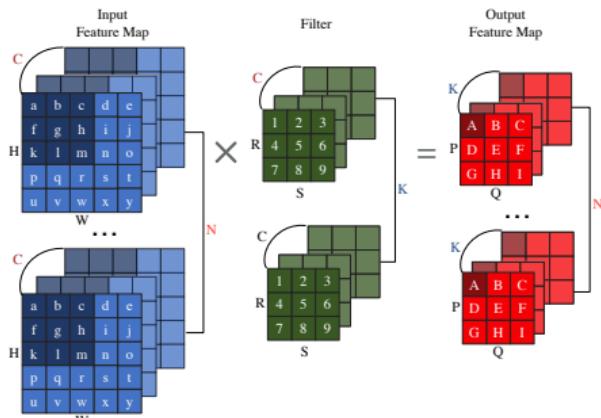
```
1  for (n=0; n<N; n++) {  
2      for (r=0; r<R; r++) {  
3          for (s=0; s<S; s++) {  
4              for (c=0; c<C; c++) {  
5                  for (k=0; k<K; k++) {  
6                      float curr_w = W[r][s][c][k];  
7                      for (p=0; p<P; p++) {  
8                          for (q=0; q<Q; q++) {  
9                              h = p * stride - pad + r;  
10                             w = q * stride - pad + s;  
11                             OA[n][k][p][q] += IA[n][c][h][w] * curr_w;  
12 } } } } } }
```

Direct Convolution: Loop Ordering + Unrolling



```
1   for (n=0; n<N; n++) {
2     for (r=0; r<R; r++) {
3       for (s=0; s<S; s++) {
4         spatial_for (c=0; c<C; c++) {
5           spatial_for (k=0; k<K; k++) {
6             float curr_w = W[r][s][c][k];
7             for (p=0; p<P; p++) {
8               for (q=0; q<Q; q++) {
9                 h = p * stride - pad + r;
10                w = q * stride - pad + s;
11                OA[n][k][p][q] += IA[n][c][h][w] * curr_w;
12             } } } } }
```

Direct Convolution: Loop Ordering + Unrolling + Tiling



```
1   for (n=0; n<N; n++) {  
2     for (r=0; r<R; r++) {  
3       for (s=0; s<S; s++) {  
4         for (c_t=0; c_t<C/16; c_t++) {  
5           for (k_t=0; k_t<K/64; k_t++) {  
6             spatial_for (c_s=0; c_s<16; c_s++) {  
7               spatial_for (k_s=0; k_s<64; k_s++) {  
8                 int curr_c = c_t * 16 + c_s;  
9                 int curr_k = k_t * 64 + k_s;  
10                float curr_w = W[r][s][curr_c][curr_k];  
11                for (p=0; p<P; p++) for (q=0; q<Q; q++) {  
12                  h = p * stride - pad + r; w = q * stride - pad + s;  
13                  OA[n][curr_k][p][q] += IA[n][curr_c][h][w] * curr_w;  
14                } } } } } }
```

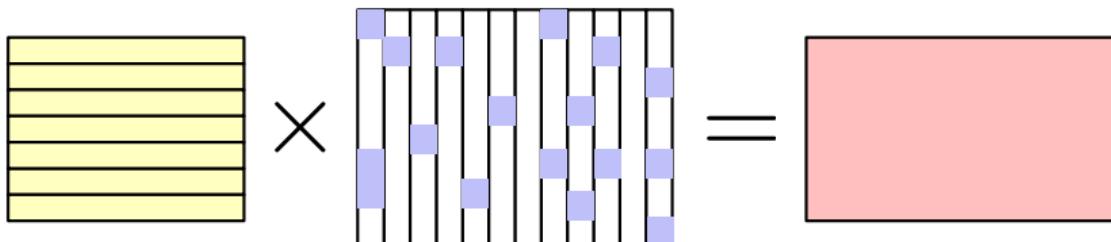


Sparse Convolution

Sparse Convolution



- Our DNN may be **redundant**, and sometimes the filters may be **sparse**
- Sparsity can be helpful to overcome **over-fitting**



Sparse Convolution: Naive Implementation 1



$$\begin{matrix} X & \quad \\ \begin{array}{|c|c|c|c|}\hline 0 & 0 & 3 & 0 \\ \hline 7 & 0 & 0 & 0 \\ \hline 0 & 0 & 4 & 8 \\ \hline 6 & 5 & 3 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 8 \\ \hline \end{array} & \begin{array}{l} * \\ \begin{array}{|c|}\hline w \\ \hline 0 \\ 0 \\ 4 \\ 8 \\ \hline \end{array} \end{array} \end{matrix}$$

Algorithm 1 Sparse Convolution Naive 1

```
1: for all  $w[i]$  do
2:   if  $w[i] = 0$  then
3:     Continue;
4:   end if
5:   output feature map  $Y \leftarrow X \times w[i];$ 
6: end for
```



$$\begin{array}{c} X \\ \hline
 \begin{array}{|c|c|c|c|} \hline 0 & 0 & 3 & 0 \\ \hline 7 & 0 & 0 & 0 \\ \hline 0 & 0 & 4 & 8 \\ \hline 6 & 5 & 3 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 8 \\ \hline \end{array} * \begin{array}{|c|} \hline w \\ \hline \begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline 4 \\ \hline 8 \\ \hline \end{array} \end{array}
 \end{array}$$

Algorithm 2 Sparse Convolution Naive 1

```

1: for all  $w[i]$  do
2:   if  $w[i] = 0$  then
3:     Continue;
4:   end if
5:   output feature map  $Y \leftarrow X \times w[i];$ 
6: end for

```

BAD implementation for Pipeline!

Instr. No.	Pipeline Stage							
	IF	ID	EX	MEM	WB			
1								
2			EX	MEM	WB			
3			IF	ID	EX	MEM	WB	
4				IF	ID	EX	MEM	
5					IF	ID	EX	
Clock Cycle	1	2	3	4	5	6	7	

Sparse Matrix Representation



A

0	0	3	0
7	0	0	0
0	0	4	8
6	5	3	0
2	0	0	1
0	0	0	8

A matrix example

rowptr

- row0 (3,2)
- row1 (7,0)
- row2 (4,2), (8,3)
- row3 (6,0), (5,1), (3,2)
- row4 (2,0), (1,3)
- row5 (8,3)

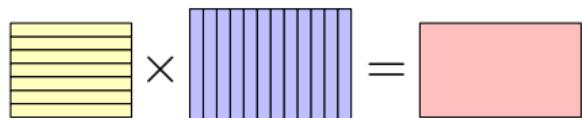
Compressed
Sparse Row
(CSR)

colptr

- col0 (7,1), (6,3), (2,4)
- col1 (5,3)
- col2 (3,0), (4,2), (3,3)
- col3 (8,2), (1,4), (8,5)

Compressed
Sparse Column
(CSC)

- CSR: Good for operation on **feature maps**
- CSC: Good for operation on **filters**
- We have better control on filters, thus usually CSC.



Sparse Convolution: Naive Implementation 2



matrix * sparse vector

$$\begin{array}{c} \text{X} \\ \boxed{0 \ 0 \ 3 \ 0} \\ \boxed{7 \ 0 \ 0 \ 0} \\ \boxed{0 \ 0 \ 4 \ 8} \\ \boxed{6 \ 5 \ 3 \ 0} \\ \boxed{2 \ 0 \ 0 \ 1} \\ \boxed{0 \ 0 \ 0 \ 8} \end{array} * \begin{array}{c} \text{W} \\ \boxed{0} \\ \boxed{0} \\ \boxed{4} \\ \boxed{8} \end{array} = \begin{array}{c} \text{Y} \\ 12 \\ 0 \\ 16 \\ 12 \\ 0 \\ 0 \end{array}$$

$$\begin{array}{c} \text{X} \\ \boxed{0 \ 0 \ 3 \ 0} \\ \boxed{7 \ 0 \ 0 \ 0} \\ \boxed{0 \ 0 \ 4 \ 8} \\ \boxed{6 \ 5 \ 3 \ 0} \\ \boxed{2 \ 0 \ 0 \ 1} \\ \boxed{0 \ 0 \ 0 \ 8} \end{array} * \begin{array}{c} \text{W} \\ \boxed{0} \\ \boxed{0} \\ \boxed{4} \\ \boxed{8} \end{array} = \begin{array}{c} \text{Y} \\ 12 \\ 0 \\ 80 \\ 12 \\ 8 \\ 64 \end{array}$$

- **BAD** implementation for Spatial Locality!
- **Poor** memory access patterns

SOTA 2: Sparse Convolution

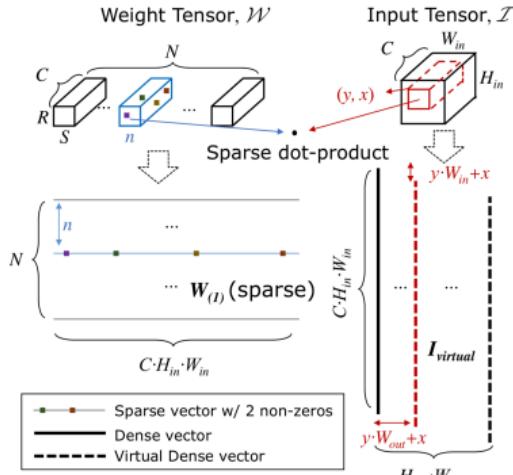


Figure 1: Conceptual view of the direct sparse convolution algorithm. Computation of output value at (y, x) th position of n th output channel is highlighted.

```

for each output channel n {
    for j in [W.rowptr[n], W.rowptr[n+1]) {
        off = W.colidx[j]; coeff = W.value[j]
        for (int y = 0; y < H_OUT; ++y) {
            for (int x = 0; x < W_OUT; ++x) {
                out[n][y][x] += coeff*in[off+f(0,y,x)]
            }
        }
    }
}

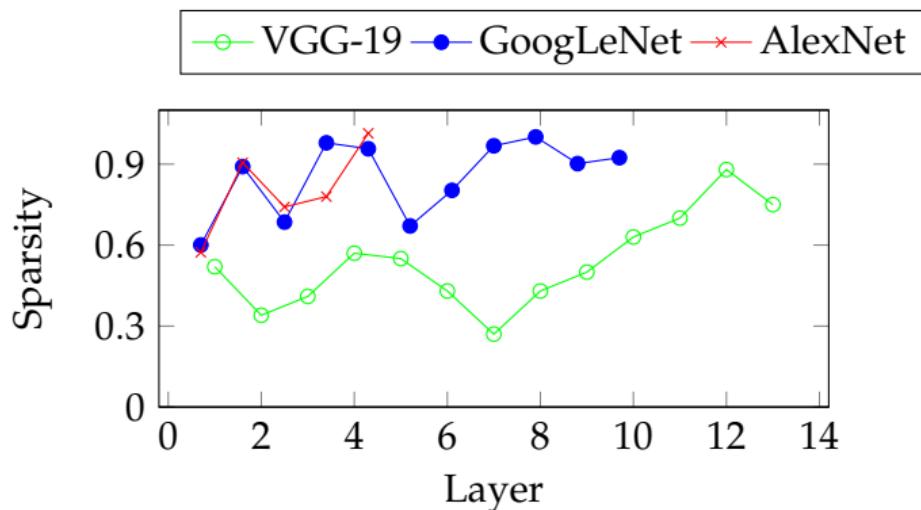
```

Figure 2: Sparse convolution pseudo code. Matrix \mathbf{W} has *compressed sparse row* (CSR) format, where $\text{rowptr}[n]$ points to the first non-zero weight of n th output channel. For the j th non-zero weight at (n, c, r, s) , $\text{W.colidx}[j]$ contains the offset to (c, r, s) th element of tensor in , which is pre-computed by layout function as $f(c, r, s)$. If in has CHW format, $f(c, r, s) = (cH_{in} + r)W_{in} + s$. The “virtual” dense matrix is formed on-the-fly by shifting in by $(0, y, x)$.

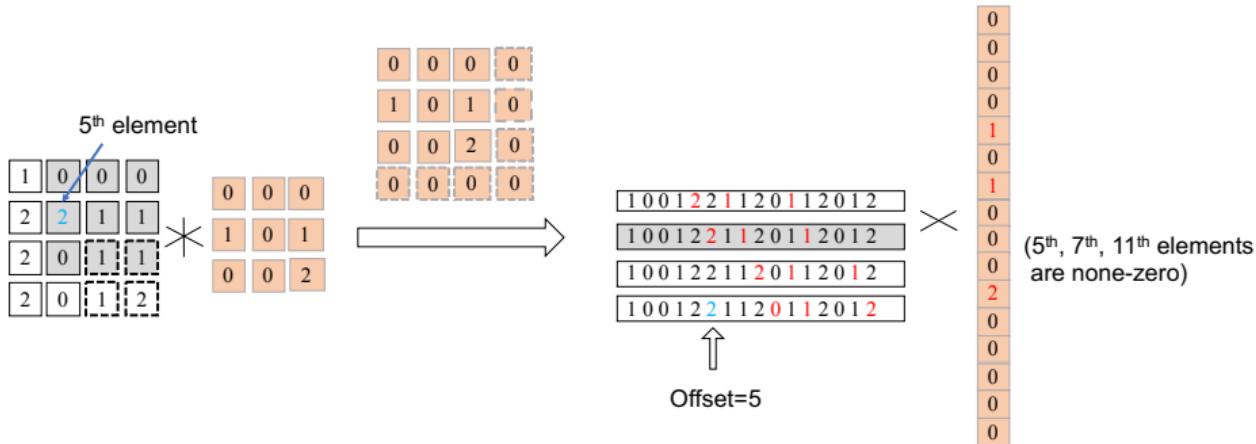
Discussion: Sparse-Sparse Convolution



- Sparsity is a desired property for computation acceleration. (cuSPARSE library, direct sparse convolution, etc.)
- Sometimes not only the filters but also the **input feature maps** are sparse.



Discussion: Sparse-Sparse Convolution



- Efficient programming implementation required; (**Improve pipeline efficiency**)
 - When sparsity(*input*) = 0.9, sparsity(*weight*) = 0.8, more than **10×** speedup;
 - Some other issues:
 - How to be compatible with pooling layer?
 - Transform between dense & sparse formats